UDC 004.05

A Method to Verify Parallel and Distributed Software in C# by Doing Roslyn AST Transformation to a Promela Model

Sergey Staroletov and Anatoliy Dubko

(Polzunov Altai State Technical University)

In this paper, we describe an approach to formal verification of parallel and distributive programs in C#. We use Microsoft Roslyn technique to get syntax and semantic information about interesting constructions in the real source code to generate some corresponding code in Promela language, designed to model actor-based interoperation systems, so we do a program-to-model transformation. Then, we verify the usual problems of parallel and distributive code by checking pre-defined LTL formulas for the model program. We are able to provide checking of data races, improper locking usage, possible deadlocks in distributive service interoperations using the Model Checking approach. This method can be used to construct a static analyzer for the .NET platform.

Keywords: *Roslyn, Verification, Static Analyzer, LTL, SPIN*

1. Introduction

This work is dedicated to improving the quality of modern software which has parallel executable entities and acts as a distributive system or microservice. Such kind of systems can have tricky errors, just exposed in rare situations. It is usually impossible or very challenging to detect such errors by testing.

Formal verification methods were introduced to ensure the correctness and reliability of such type of program systems, to detect faults at the different stages of software development and maintenance to consistently reduce them.

We assume that the formal verification approach [1] should be applied here but it will be hard to understand by an ordinal software developer how to create different kinds of models to verify [2], so such techniques should be a transparent part of developing process, and additional checking should be integrated into a compiler or an IDE.

The problems of verification and creation of a verifying compiler are examples of the fundamental problems of modern programming that are in the progress of being solved.

In this paper, we deal with C# parallel programs and WCF services. We use Microsoft Roslyn technique to obtain the AST (Abstract Syntax Tree) from C# input sources to generate a corresponding code in Promela modeling language intended to make further verification of the generated model code using SPIN verifier according to defined classes of possible parallel and distributive errors and generated requirements as LTL (Linear Temporal Logic) formulas.

2. Related Work

2.1. .NET and C# for parallel and distributed systems creation

C# is an industrial, type-safe, object-oriented modern language designed to develop applications running in the .NET Framework environment [3]. Using C#, developers can create general-purpose software including standard and universal desktop applications, mobile applications, web services, distributed components; client-server, database and web applications.

The C# programs run within the .NET Framework – an integrated Windows component that contains the Common Language Runtime (CLR) virtual system and a unified set of Base Class Libraries (BCL). The CLR is an implementation of the Common Language Infrastructure (CLI) – an international standard, the foundation of execution and development environments with close interaction of languages and libraries. Novel platform implementation named .NET Core provides the developers with some abilities to create application not only for Windows platform, but also for Linux and Mac, and this implementation is even open-sourced [4].

Source code written in C# is compiled into the Intermediate Language (IL) following the CLI specification. IL code and resources, such as bitmaps and strings, are stored on disk in an executable file, called an assembly, with EXE or DLL extension.

The .NET platform and the C# language provide the ability to create distributed components or applications:

- Web API. The Web API is a RESTful HTTP web service that can interact with various components. These can be ASP.NET web-, mobile or regular desktop applications.
- WCF (Windows Communication Foundation). WCF provides a platform for building service-oriented (SOA) applications and implements a manageable approach to create web services and clients for them [5]. With WCF, developers can send data in the form of asynchronous messages from one service endpoint to another. The endpoint can be a part of a permanently available service hosted in IIS (a web-based Internet Information Services server) or represent a service hosted inside an application. Messages can be in

the form of a complex stream of binary data.

- .NET Sockets. Sockets are used to build traditional client-server applications. By connecting two sockets explicitly, the applications can transfer data between different processes, nodes or platforms.

In the current work, we model distributive interoperations only as WCF services and clients due to their high-level logical structure.

2.2. Roslyn

Roslyn [7] is a platform that provides the system developers with various powerful tools for analyzing and parsing .NET languages (mostly C#) code. The source code of this platform is freely available on MS GitHub account [6].

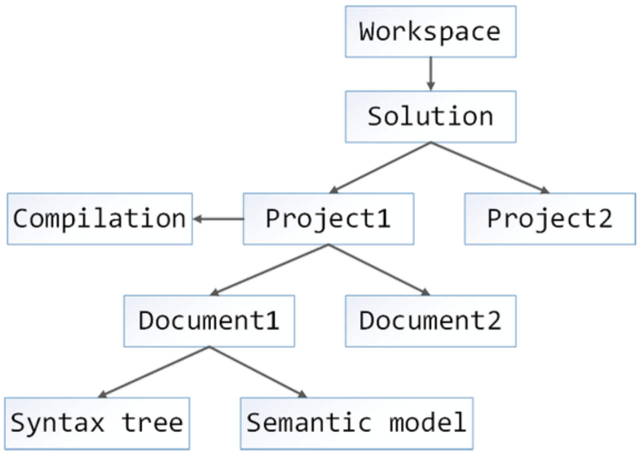


Fig. 1. Retrieving data for analysis with Roslyn [8]

With the help of tools provided by the Roslyn platform, it is possible to perform a full syntax analyzes of the code by traversing all supported language constructs. The Visual Studio environment allows developers to create tools embedded in the IDE itself (as Visual Studio extensions) and independent applications on the basis of Roslyn.

When analyzing code with Roslyn tools, it is possible to get a list of files from a solution whose source code is being checked, to get the necessary entities for parsing (syntax model), and then to get access to the semantic model of the program after compiling it (Figure 1).

For a complicated analysis, it is necessary to obtain a syntax tree and a semantic model. A syntax tree is built from a program source code and is used to link various language constructs. A semantic model provides information about program objects and their types.

For every language structure, Roslyn defines corresponding type nodes. Moreover, for each

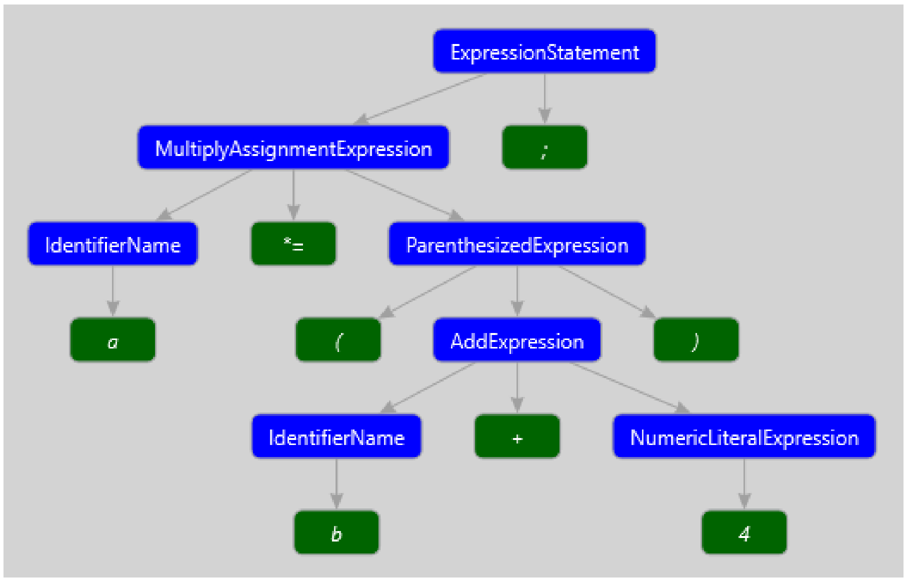


Fig. 2. An expression tree for a * = (b + 4) with lexemes [8]

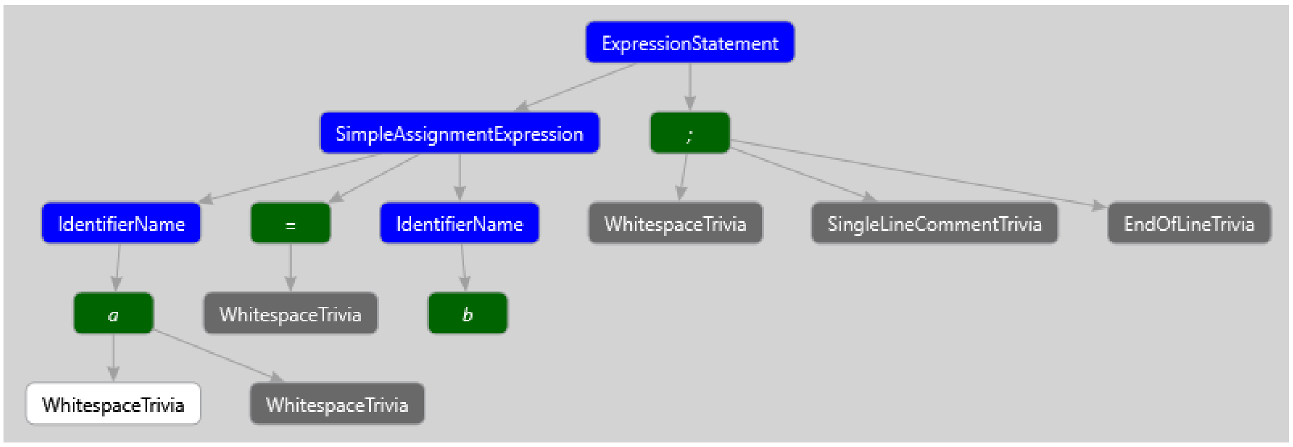


Fig. 3. A sintax trivia tree for $a = b; //Comment$ [8]

node type, a method in code can be defined that performs a crawl through the nodes of this type. Thus, by adding handlers to traverse of various nodes, we can analyze only the constructions of language we are interested in. An example tree for the expression $a* = (b + 4)$ in syntax tokens is shown in Figure 2.

In addition, there exist "syntax trivia" – elements in the tree that will not be compiled into IL code, they are stands for some additional syntax information. This category includes comments, preprocessor directives, spaces, etc. Figure 3 shows a tree with extra syntax information for the expression $a = b; //Comment$. This data can be possibly used for implementing additional processing for constructions as special comments, for example, it is a usual way to write control annotations for deductive verification approach now, as ACSL annotations for C programs to prove with Frama-C tool are written in C comments [9].

The semantic model in Roslyn provides useful information about objects and their types. It is a powerful source for building deep and complex analyzers. That is why it is essential to have a correct compilation and a correct semantic model.

2.3. Actor model

Today, the dominant way to programs creation is the imperative approach based on the general state (it is also used for C# programs). Very often, the code of such programs from the very beginning is written without considering the possibility of parallelization, and parallel actions occur in the code only when needed.

The actor model on the other side, "forces" the code to be parallel from the starting point. This model is a mathematical representation of parallel computing, which uses the concept of "actor" as a universal primitive. It was applied [10] as a basis for understanding the calculation using processes and as a theoretical basis for some practical implementations of parallel systems.

The basic idea is that the actor-based application is built from many lightweight processes called *actors* [11]. Each actor is responsible for one tiny task, so it is easy to understand what does it responsible for. Programs with more complex logic can be implemented as an interaction of several actors that are concurrently sending messages between each other.

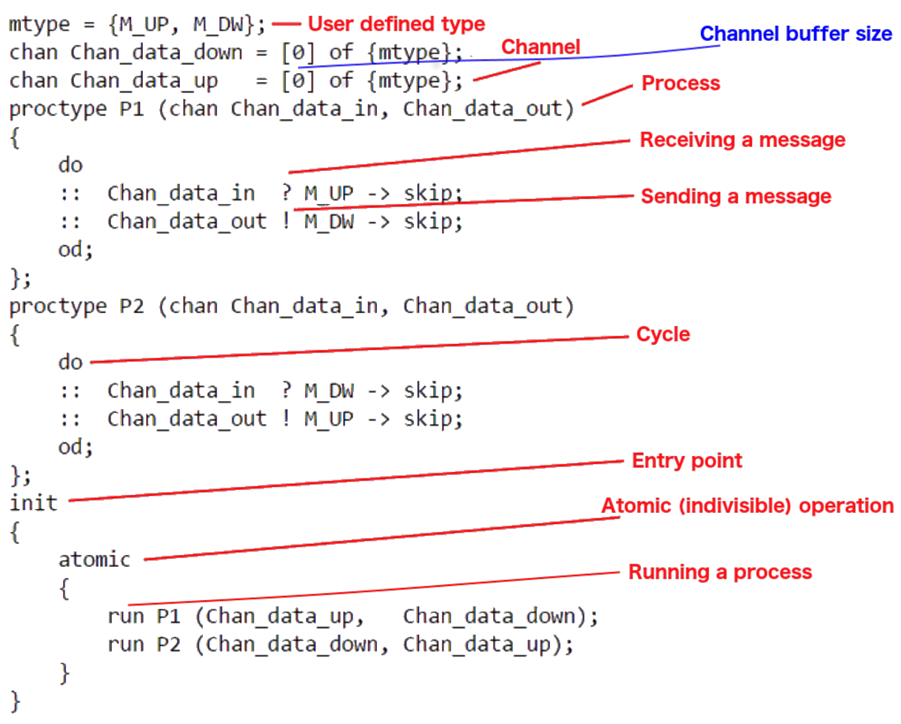
So, the actor is a computational entity that, in response to a received message, can simultaneously:

- send a finite number of messages to other actors;
- create a finite number of new actors;
- select the behavior that will be used when processing the next received message.

It does not assume the existence of a specific sequence of the above-described actions and all of them can be performed in parallel. Separating the sender from the messages was a fundamental achievement of the actor model. Message recipients are identified by the address (PID, process identifier), which is sometimes called the mailbox address. Thus, an actor can interact only with those actors whose addresses it has, and it can extract addresses from received messages or know them in advance.

The most outstanding implementation of the actor model was made in Erlang language [12]. A rather popular implementation of this model is Akka library [13] for Scala.

2.4. Promela and SPIN



```
mtype = {M_UP, M_DW};
chan Chan_data_down = [0] of {mtype};
chan Chan_data_up   = [0] of {mtype};
proctype P1 (chan Chan_data_in, Chan_data_out)
{
    do
    ::  Chan_data_in  ? M_UP -> skip;
    ::  Chan_data_out ! M_DW -> skip;
    od;
};
proctype P2 (chan Chan_data_in, Chan_data_out)
{
    do
    ::  Chan_data_in  ? M_DW -> skip;
    ::  Chan_data_out ! M_UP -> skip;
    od;
};
init
{
    atomic
    {
        run P1 (Chan_data_up,   Chan_data_down);
        run P2 (Chan_data_down, Chan_data_up);
    }
}
```

Fig. 4. An example Promela model

SPIN [14] is a utility for verifying the correctness of distributed software models. The abbreviation stands for Simple Promela INterpreter. This utility is used for automated verification of models, and it can also work as a simulator, executing one of the possible traces of a model of system behavior.

The SPIN system checks *not the programs themselves, but their models.* To build a model for an original parallel program or an algorithm, the engineer (usually manually) builds a representation of this program in the C-like input language, called Promela (Protocol MEta-LAnguage). This program in Promela language can be considered a model of the verified program. Promela language constructs are simple, they have clear and distinct semantics, which allows translating any program in this language into a transition system with a finite number of states for verification purposes. The requirements for the model are expressed in the language of LTL (Linear-time Temporal Logic) [15].

Input models in Promela are different from original verifiable programs, usually written in high-level programming languages. Promela programs do not have classes, and they represent a flat structure of interacting parallel processes, as we described in the actor model section (Figure 4), have a minimum of control structures, all variables have finite domains. Therefore, such a program can be considered a model of the system being analyzed; it represents an abstraction of the original system, in which the engineer should reflect those aspects and characteristics of the real system that are very significant for the properties specified for the verification.

The system description is expressed in Promela language must preserve the essential properties of this system. It should be stressed that the resulting verification quality of Promela programs entirely depends on the degree of adequacy of the constructed model. The model construction process can be done manually or automatically, and in this paper, we present the ways of auto model generation based on the C# compiler information.

2.5. Existing solutions to do C# code verification

To check the C# programs statically, Microsoft Research offered a solution called Spec# (Specification Sharp) which extends C# language with constructs for non-null types, preconditions, postconditions, and object invariants [17]. With this solution, the developer should manually specify additional code to describe the program with special requirements in the form of logical predicates. Later, in [18] an embedded code contracts approach was presented, it became a part of the .NET platform, introducing annotations to the C# classes and ways to statically check the contracts assumptions while code writing and compiling from the IDE. It is intended to prove the program logic, and it is hard to check the interoperations with this approach.

MS Research has some trying to create a formal language to describe models with message passing, and in [19] Sing# language was presented as an extension to Spec# but the current state of the project is unclear and it seems they stopped developing and using it.

In [20] ISP RAS introduced an approach to do static analysis of C# programs based on the symbolic execution method with using advanced SMT solvers.

In [21] the PVS-Studio static analyzer for C, C++ and C# programs was described and its internal methods were discussed. It can detect some threading issues, and it uses Roslyn as a backend.

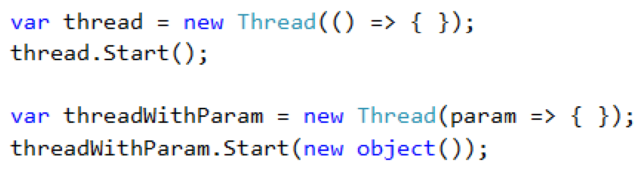
We can state that none of the described tools uses Promela and SPIN as a way to check the extracted models from C# input code. The analyzers can use Roslyn to obtain AST, but then they use own special techniques. None of the methods can verify distributed service interoperations.

3. Code Analysis and Model Generation

We consider here some algorithms for transforming syntax elements we are interested in the C# programs, to a Promela code according to our goals. Some ideas of it were given in our paper [22].

3.1. Ways of thread creation and its modeling

Consider some common ways to create parallel threads in C# language. Firstly, a thread can be created with the Thread class from the System.Threading namespace (Figure 5).

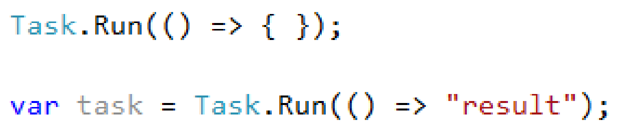


```
var thread = new Thread(() => { });
thread.Start();

var threadWithParam = new Thread(param => { });
threadWithParam.Start(new object());
```

Fig. 5. Creating Threads with the Thread Class

This class is used to create two types of threads: parameterized and non-parameterized. Its constructor takes a delegate of type ThreadingStart or ParametrizedThreadingStart, explicitly or implicitly. In Figure 5, the parameter is passed as a lambda expression, which is implicitly reduced to the above types.

Also, a parallel code can run with the Task class from the System.Threading.Tasks namespace (Figure 6).

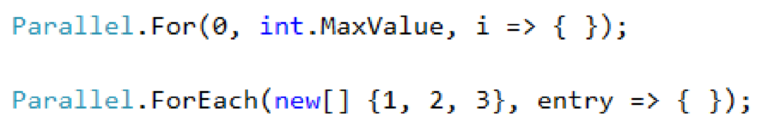


```
Task.Run(() => { });

var task = Task.Run(() => "result");
```

Fig. 6. Creating Threads with the Task Class

Using classes from this namespace, one can create high-level thread types, taking advantage of thread-pooling. The static Run method accepts an Action type delegate with no parameter or Func delegate (from the System namespace), which can return a result. The method returns its result in the form of a new object of type Task, which represents the running task passed as the parameter.

Another method is the Parallel class from the System.Threading.Tasks namespace (Figure 7).



```
Parallel.For(0, int.MaxValue, i => { });

Parallel.ForEach(new[] {1, 2, 3}, entry => { });
```

Fig. 7. Creating Threads with the Parallel Class

Also, we consider some methods are used to create parallel loop processing. The static For method takes as its parameters the initial value of the counter, the final value of the counter, and a parameterised delegate (of type Action) that handles the current value of the counter.

The static ForEach method takes as its parameters a collection that implements the typed IEnumerable <T> interface and a parameterized delegate (of type Action) that accepts the current collection element in the iteration.

Now consider modeling the interaction of threads in C# as Promela structures. For the simulation, we decided to use a separate Promela process for each running thread in C#. The interaction between threads, as well as the awaiting of their completion, are modeled by transmitting synchronization messages through Promela channels. The types of processes are described using the declaration with the *proctype* keyword [16]. Processes are always declared globally. Processes are started from other processes by the means of the *run* operator.

Later we describe a way to the code generation for this body from all the possible variants of parallel entity creation described in this section.

3.2. Analysis of the syntax, semantics and data flow of the C# programs and its modeling

Before starting the analysis of C# sources, it is necessary to construct a graph of method calls inside the program being analyzed. This graph will help:

- Firstly, detect calls of methods that trigger a new thread.
- Then determine the order of calls, starting from an arbitrary method to generate definitions in Promela before their use further.
- And as a result, switch to a pure interprocess communication model in Promela without calling intermediate methods.

Consider an example of a method call in C# (Figure 8).

We see, in order to find all method calls in the source code of the program, we need to find all nodes of the InvocationExpression type in its syntax tree.

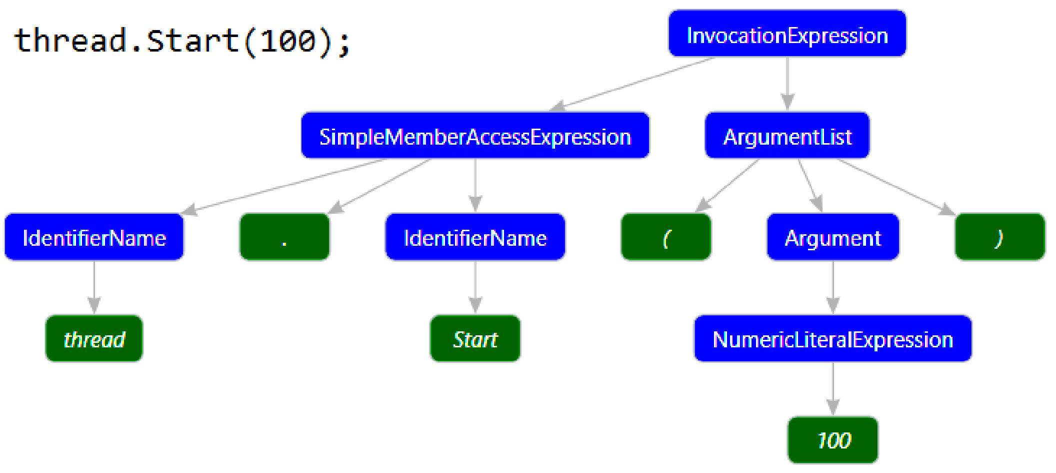


Fig. 8. Method Call syntax tree

To construct the graph, it is necessary to determine the relationship *"which method is called from a which one"*. It is required for each method call to find a definition of its parent method or a lambda expression (Figure 9).

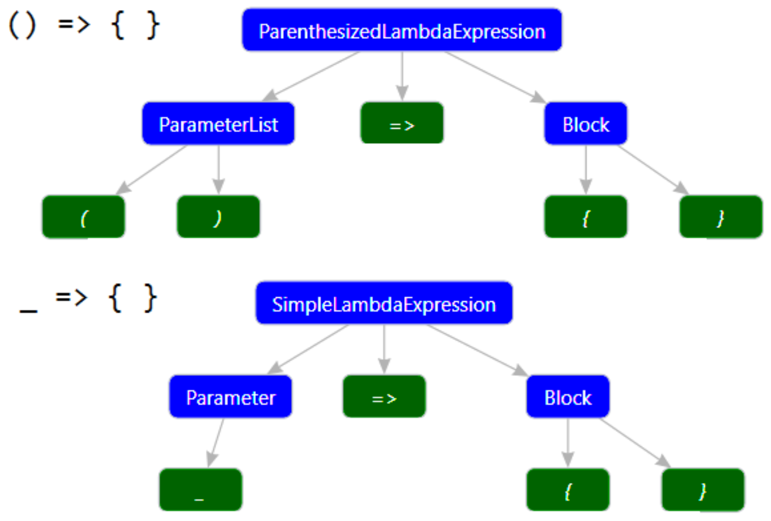


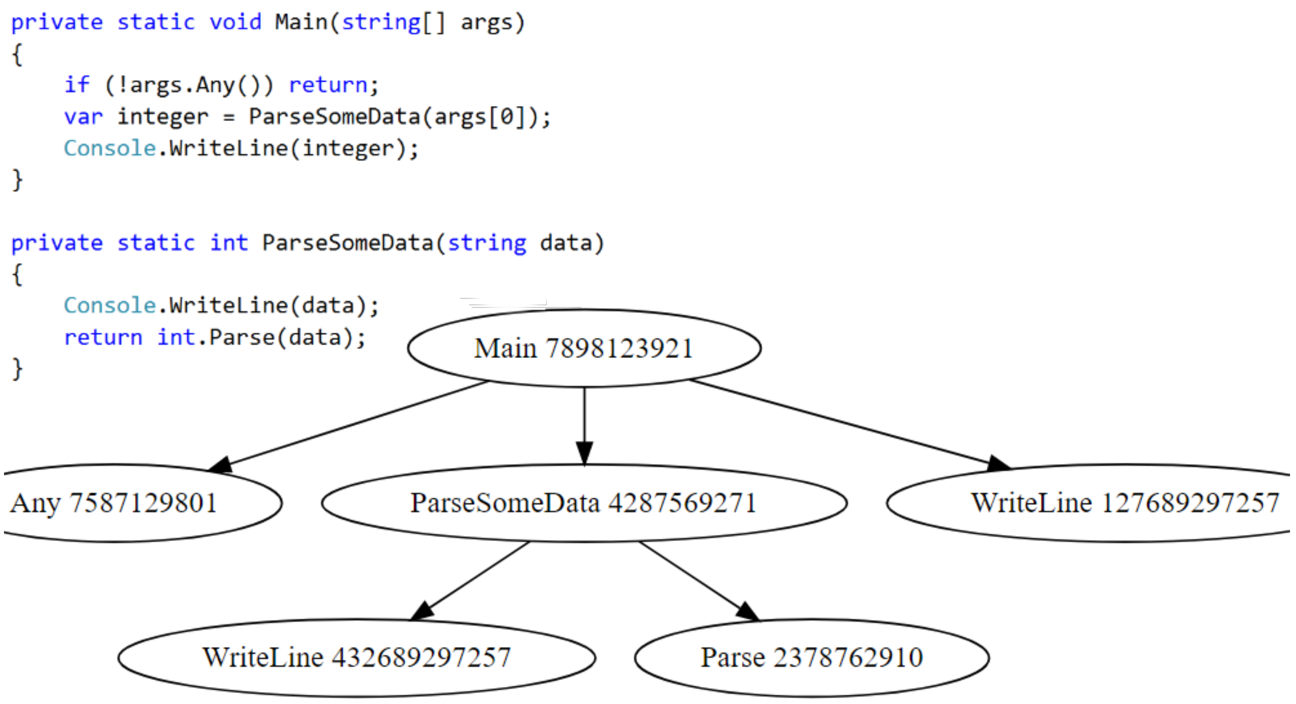
Fig. 9. Lambda Expression syntax tree

Moving up in the tree starting from a method call, we should find such a parent definition of the method or a lambda expression. This link will be an edge in the call graph.

We discovered that each node in the syntax tree has already defined hash code. Due to this, it is possible to use it as a unique value of a vertex in the graph without fear of the collisions occurrence between identical syntactic structures in the source code.

An example of the resulting call graph is shown in Figure 10.

The graph is acyclic, but it is not always connected. With further code generation on



```
private static void Main(string[] args)
{
    if (!args.Any()) return;
    var integer = ParseSomeData(args[0]);
    Console.WriteLine(integer);
}

private static int ParseSomeData(string data)
{
    Console.WriteLine(data);
    return int.Parse(data);
}
```

Fig. 10. Call graph in C# source code

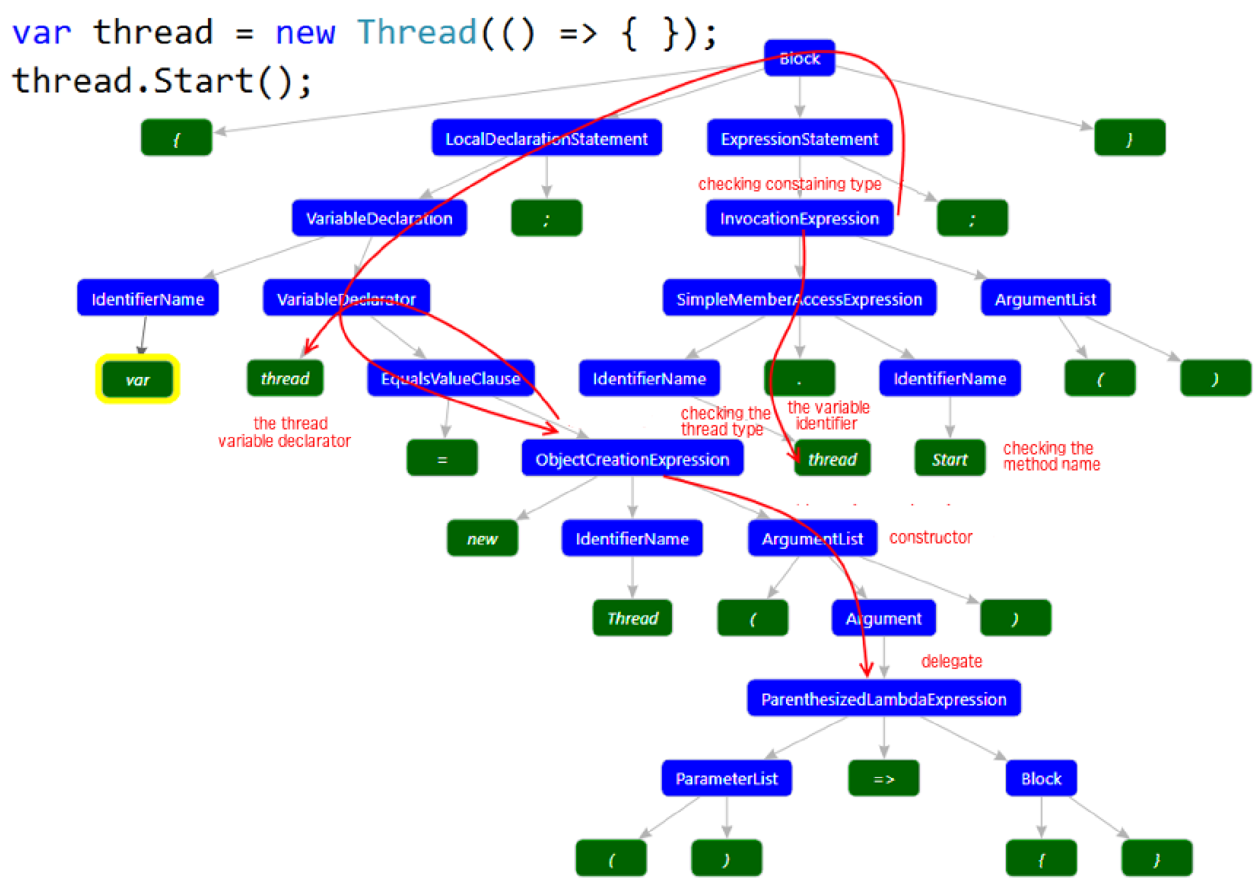
Promela, acyclicity is easily removed when using the semantic model, since in this model the same characters have the same hash codes.

During the construction of the call graph, we can immediately determine which of the calls starts a new thread. For this, it is necessary to make a semantic analysis of the method call, setting some conditions:

- The name of the method that starts the thread.
- The type of object with which the method is called.
- If the method is not static, check, was the constructor called with the delegate of this thread object.

The first two conditions are checked using the appropriate properties of the method call symbol in the semantic model. To check the last condition, it is necessary to carry out an additional analysis:

- If a static method is called to start a thread, then check its argument, which must be a delegate.
- If the method is called immediately after the creation of the object, then analyze the child nodes of the call tree to check whether the constructor with the delegate was called.
- If the reference to the object was passed to a local variable, field or class property before calling the method, then we need to analyze the data flow of the parent method body,



```
var thread = new Thread(() => { });
thread.Start();
```

Fig. 11. Creating and starting a thread through a variable

delegate or class and link the local variable and the corresponding constructor call to the delegate, if any (Figure 11).

The Roslyn platform provides a number of useful classes to work with syntactic trees. One of them is the CSharpSyntaxWalker class, which implements the Visitor Design Pattern [23]. After inherited it, we can start a traversal through all syntactic nodes of the tree, simultaneously redefining the methods of visiting nodes of various types.

The idea of generating Promela code is precisely this crawling of the entire syntax tree of C# code with pre-determined methods for generating syntax nodes we need (Figure 12).

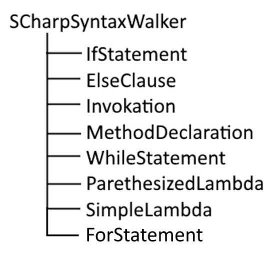


Fig. 12. C# syntax nodes where Promela code will be generated

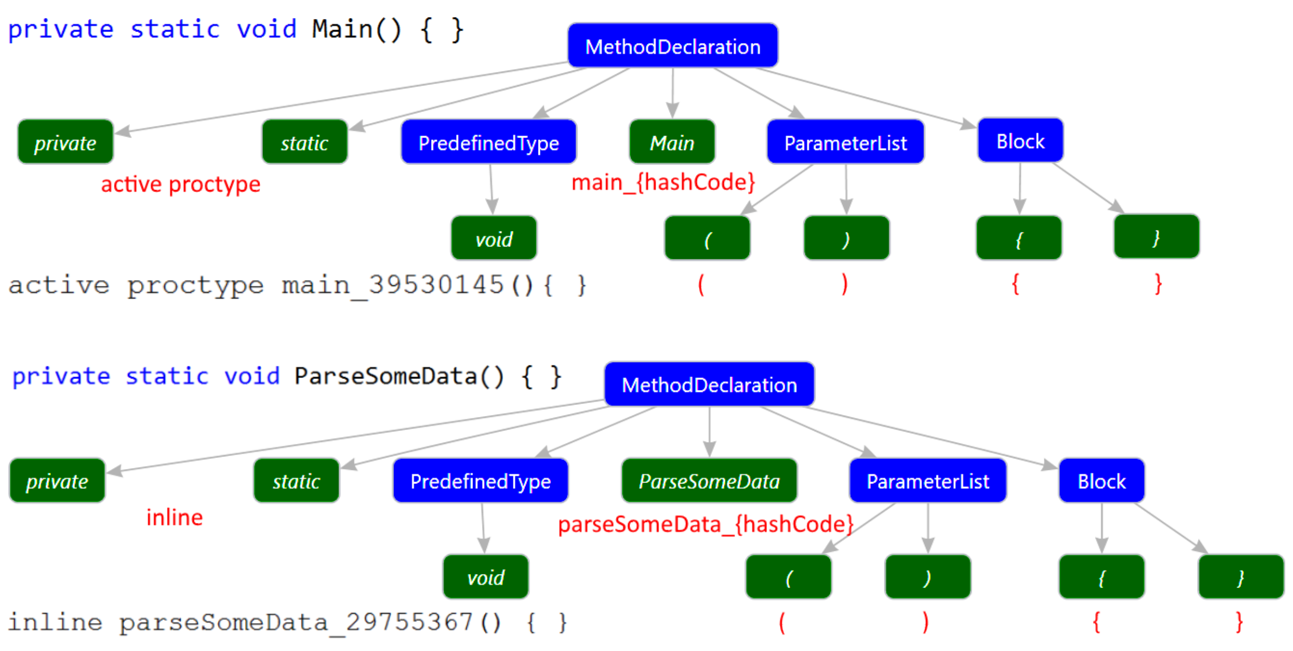


Fig. 13. Promela code generation for MethodDeclaration

Now we proceed to describe some nodes processing.

Each traversal of the MethodDeclaration, ParenthesizedLambda and SimpleLambda nodes will create a separate unit to generate Promela code, which will do this for the remaining nodes which belong to each of these declarations. Also during the crawl, an entry point to the program will be selected, and it is becoming later as the initial state when generating the resulting code using the call graph.

When visiting nodes of the MethodDeclaration type, we provide an algorithm for generating code in Promela presented in Figure 13. If the declaration is a program entry point, then it is generated as the initial running process in Promela, otherwise as an inline function [24].

When generating declarations and calls in Promela, the hash codes of corresponding symbols from the semantic model are used for names, because the same symbols, unlike syntax nodes, have the same hash codes.

When visiting nodes of ParenthesizedLambda and SimpleLambda types, which are threads delegates, a graphical illustration of this Promela code generation algorithm is shown in Figure 14. Here for each delegate, a new, not yet started process is created.

When visiting nodes of the Invocation type, a Promela code generation algorithm is provided in Figure 15. If an invocation launches a new thread, then a call code is generated for a previously generated process corresponding to the delegate. Otherwise, an inline function call is generated.

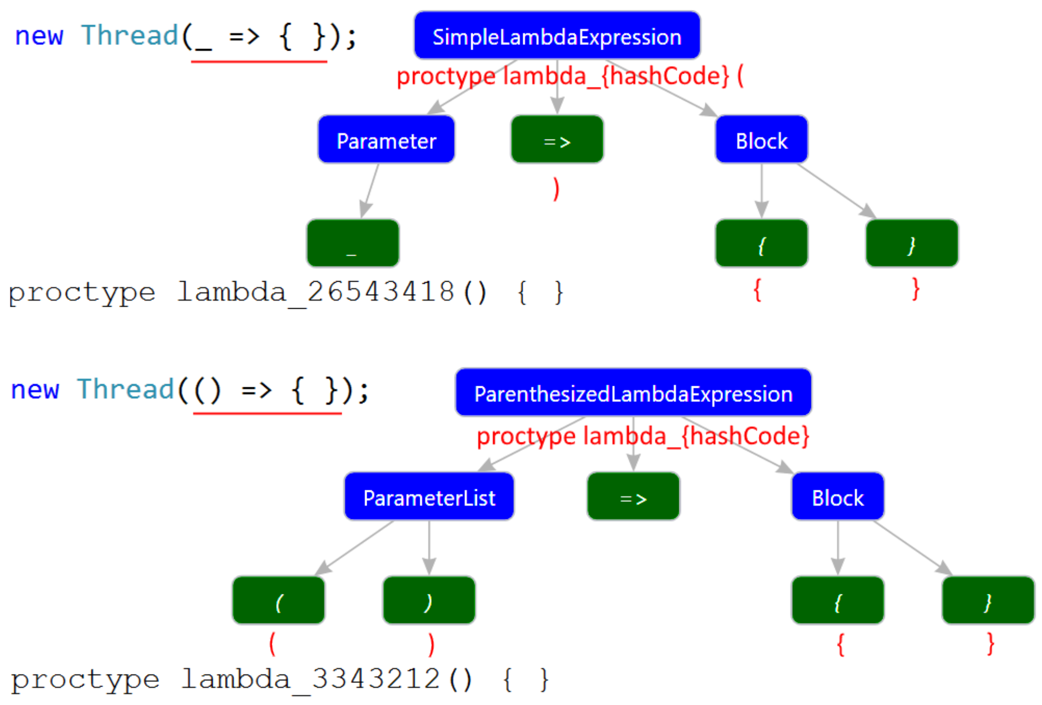
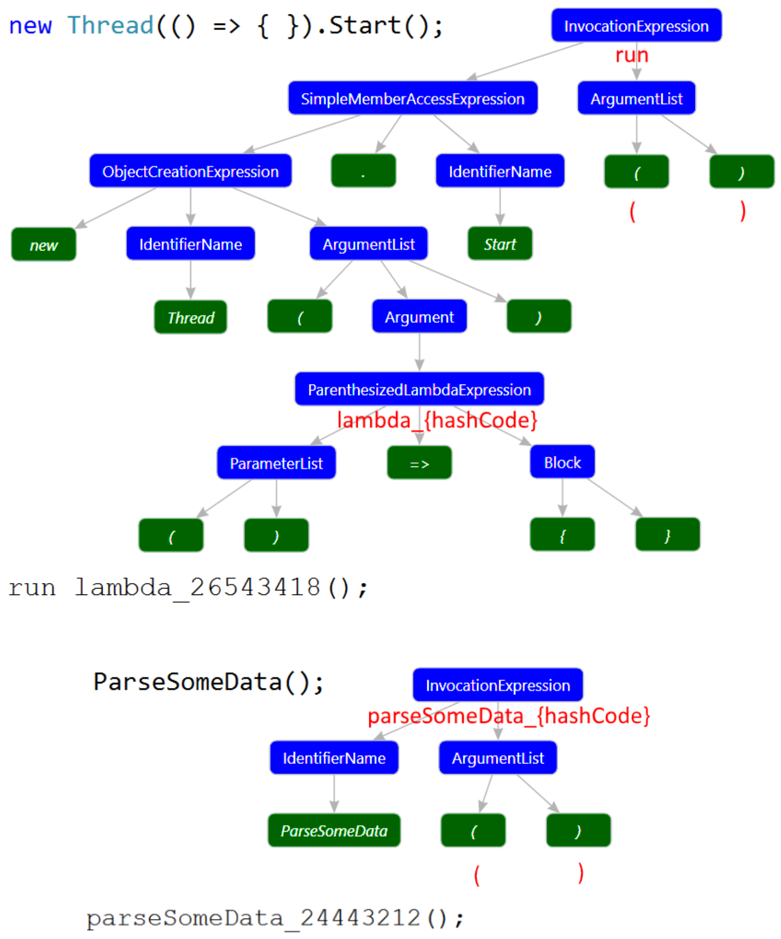

new Thread(_ => { });
SimpleLambdaExpression
proctype lambda_{hashCode} (
Parameter
=>
)
Block
_
{
}
{
}
proctype lambda_26543418() { }
new Thread(() => { });
ParenthesizedLambdaExpression
proctype lambda_{hashCode}
ParameterList
=>
Block
(
)
{
}
(
)
{
}
proctype lambda_3343212() { }

Fig. 14. Promela code generation for thread delegates

When visiting nodes like IfStatement and ElseClause, an algorithm for generating code is provided in Figure 16. A non-deterministic control flow is generated in Promela both by the if condition and by the else alternative branch, since all the variants of possible executions as steps in the control flow are necessary for further verification.

When visiting nodes of type WhileStatement, an algorithm for generating code on Promela is provided in Figure 17. A non-deterministic control flow is generated on Promela both with the do(while) clause and with the break operator, which may be absent in the C# source code since we need all variants of the possible development of program behavior and due to Promela blocking guarded condition semantic [25].



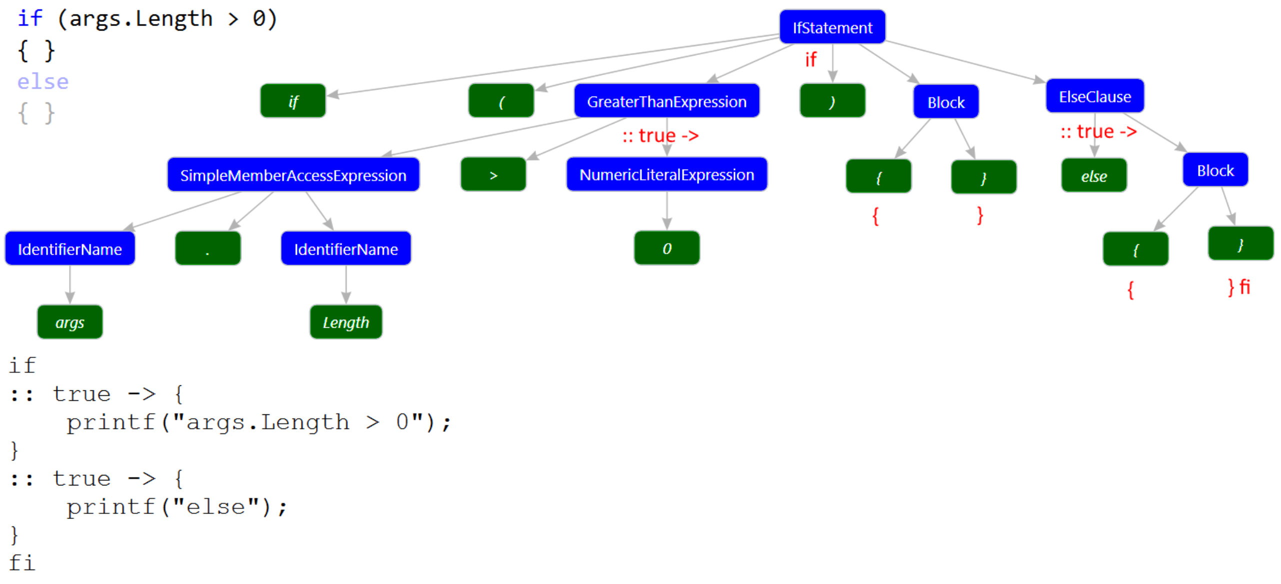
```
new Thread(() => { }).Start();
```

```
run lambda_26543418();
```

```
ParseSomeData();
```

```
parseSomeData_24443212();
```

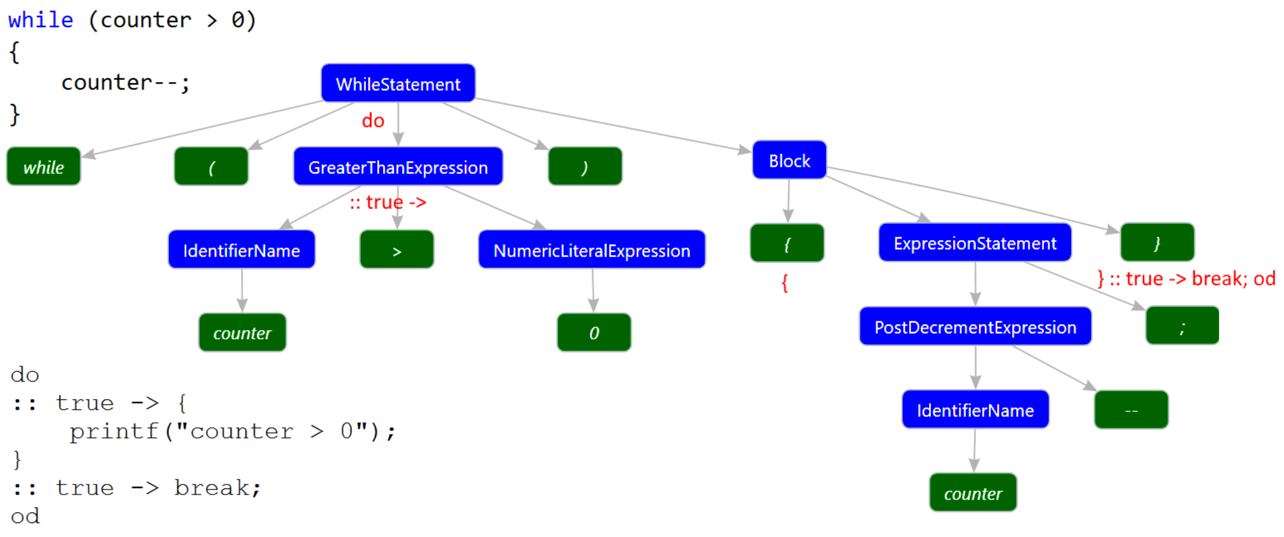
Fig. 15. Promela code generation for Invocation



```
if (args.Length > 0)
{ }
else
{ }
```

```
if
:: true -> {
    printf("args.Length > 0");
}
:: true -> {
    printf("else");
}
fi
```

Fig. 16. Promela code generation for IfStatement and ElseClause



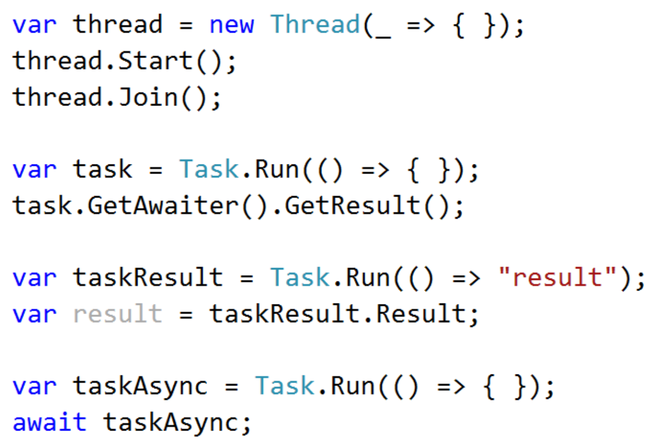
```
while (counter > 0)
{
    counter--;
}
```

```
do
:: true -> {
    printf("counter > 0");
}
:: true -> break;
od
```

Fig. 17. Promela code generation for WhileStatement

3.3. Thread waiting and its modeling

Figure 18 shows some possible ways to wait for threads termination in C# code.



```
var thread = new Thread(_ => { });
thread.Start();
thread.Join();

var task = Task.Run(() => { });
task.GetAwaiter().GetResult();

var taskResult = Task.Run(() => "result");
var result = taskResult.Result;

var taskAsync = Task.Run(() => { });
await taskAsync;
```

Fig. 18. Ways to wait for threads in C#

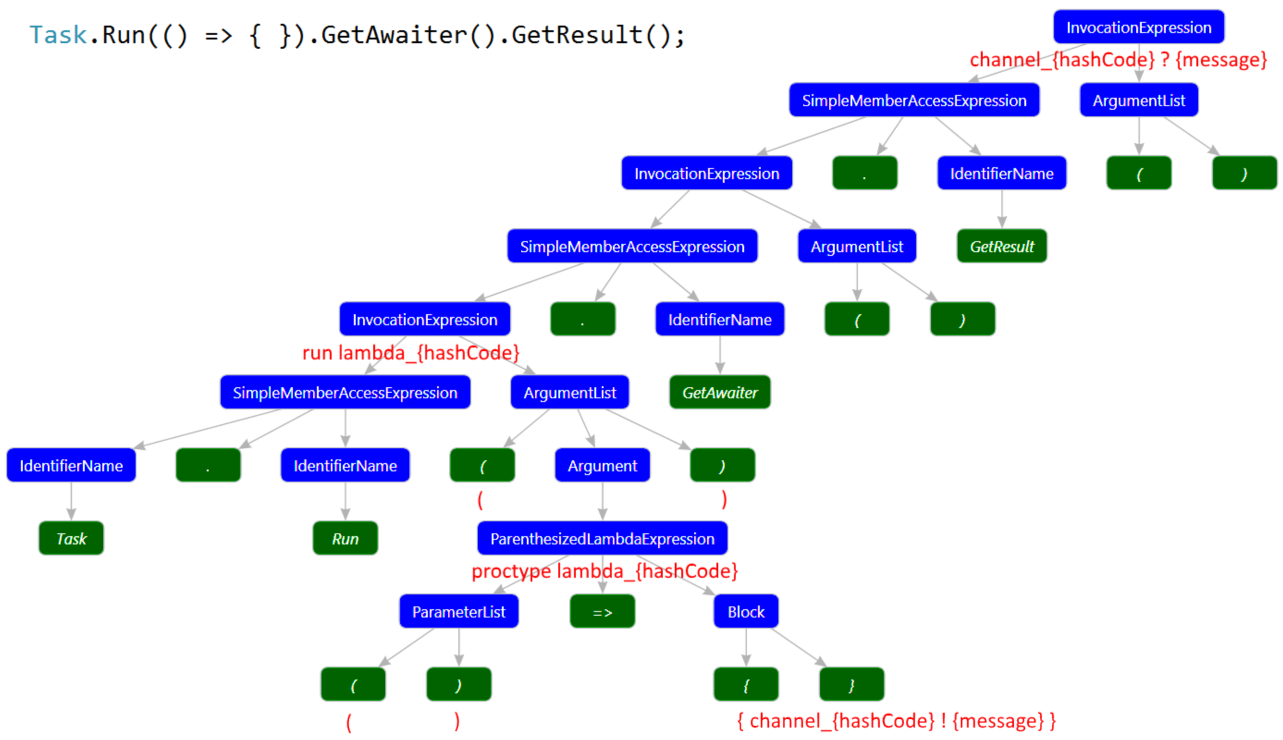
When modeling of waiting for the completion of C# threads in Promela, we use message channels. For each child task, we create a separate message channel with a buffer size of 1. This buffer size is chosen so that the child processes should be executable when sending synchronization messages at the end of their work (computation body). The channel buffer size larger than 1 is redundant, since for each child process an individual channel is created, and the buffer size equals to 0 (the rendezvous channel) will make the child process impracticable if the parent does not want to wait for its completion.

Each child process sends a message according to a specific pattern to its message channel at the end of the execution. The parent process, in turn, can stop its execution by calling the receive statement from this channel along with the same pattern. Also, this process will resume its work as soon as a message from a finished child appears on the channel.

To cover all the considered ways of waiting for threads in C#, some additional rules for syntactic nodes of the InvocationExpression type are added using the similar pattern, that rules find calls to methods which trigger a new thread. Only method names and object types here should be changed, for example, the TaskAwaiter type and the GetResult method, or the Thread type and the Join method.

For code generation, following the scheme for simulating awaiting when generating definitions of new processes, we add an appropriate channel in front of them with the same hash code as the process, and then at the end of the process body, we generate a message send to this channel (Figure 19).

```
Task.Run(() => { }).GetAwaiter().GetResult();
```

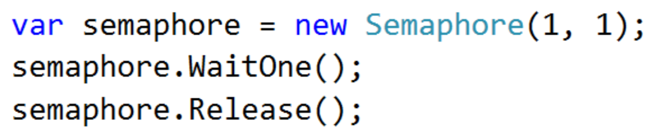


```
mtype = { thread_is_done };

chan channel_43603258 = [1] of { mtype }

proctype lambda_43603258()
{
    channel_43603258 ! thread_is_done;
}

active proctype main_39530145()
{
    run lambda_43603258();
    channel_43603258 ? thread_is_done;
}
```

Fig. 19. Promela code generation to model the thread awaiting

3.4. Blocking and its modeling

To model the blocking, we consider modeling operations with the Semaphore class due to the generality of the locking process with it. This approach can be extended to other synchronization primitives and patterns. Figure 20 shows some of the methods for accessing semaphores in C# language.



```
var semaphore = new Semaphore(1, 1);
semaphore.WaitOne();
semaphore.Release();
```

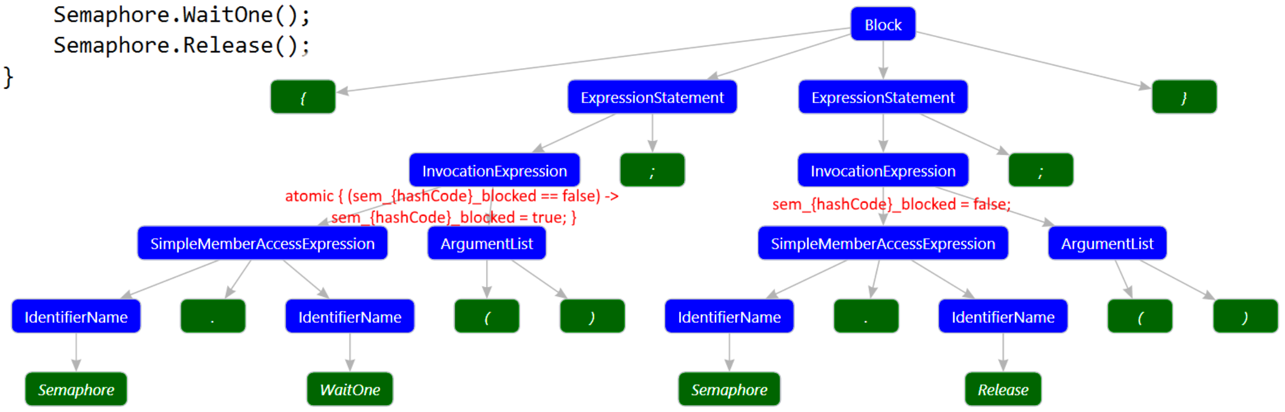
Fig. 20. Ways to create and use semaphores in C#

When modeling a semaphore, we add a shared resource, in the form of a global variable, for which wait and release actions will be modeled.

When we proceed to await and resource capture modeling, we check the resource value, and when it is considered free, then we change this value. These two steps should be concluded in one indivisible operation. For this, we use the possibility of Promela language to wrap the code into an atomic block. When we model the release of a resource, we assign the value to the variable which is considered to be free and a possible blocked code in some different process can continue to run [22] (and it is a subject to do further checks of it).

To cover the methods of interaction with semaphores in C#, additional rules are added for syntactic nodes of the InvocationExpression type using a similar pattern we did for finding calls to methods that trigger a new thread. We should change only the method name and object type: the Semaphore type, the WaitOne and Release methods (Figure 21).

```
private static readonly Semaphore Semaphore = new Semaphore(1, 1);
private static void Main(string[] args)
{
    Semaphore.WaitOne();
    Semaphore.Release();
}
```

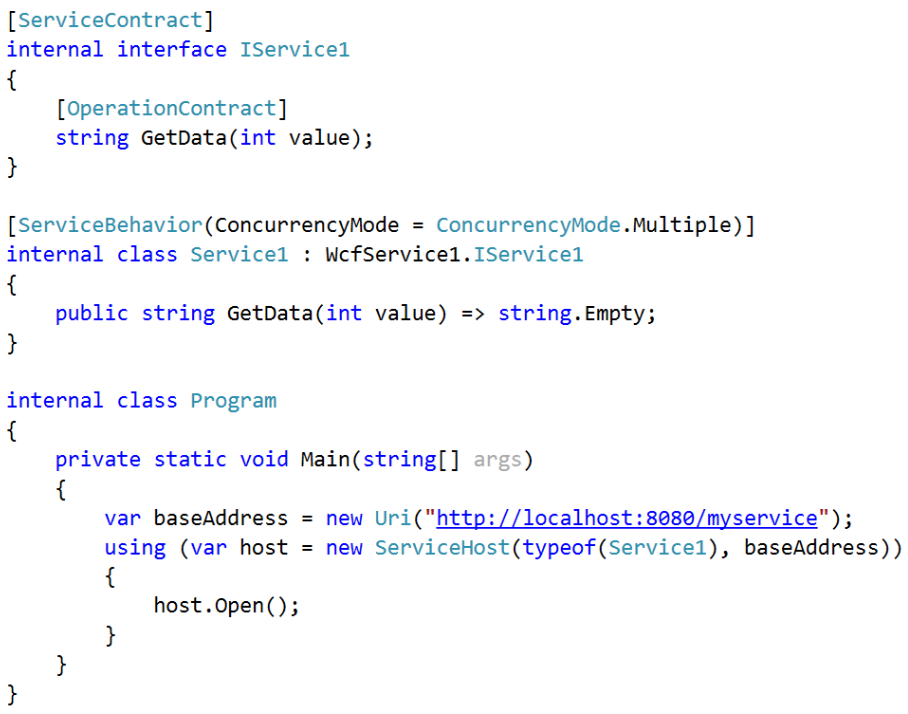


```
bool sem_63390070_blocked = false;
active proctype main_33639718()
{
    atomic {
        (sem_63390070_blocked == false) -> sem_63390070_blocked = true;
    }
    sem_63390070_blocked = false;
}
```

Fig. 21. Generating Promela code for C# Semaphore

3.5. Creating and using WCF services and its modeling

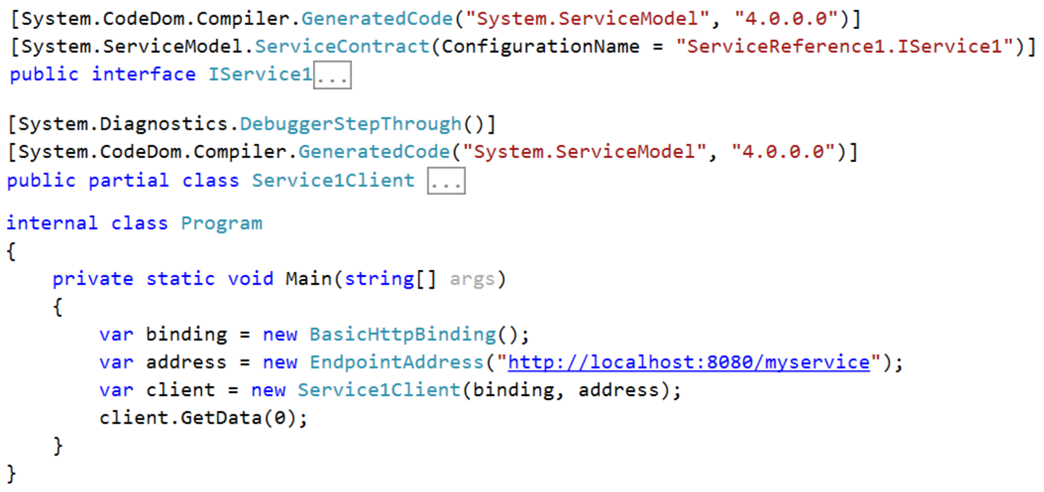
Figure 22 shows how to create and run a WCF service in C# code. The service here contains two things – a contract (an interface) and an implementation so that the service publishes the contract and clients can call its methods by using proxy-classes based on the contract interface.



```
[ServiceContract]
internal interface IService1
{
    [OperationContract]
    string GetData(int value);
}

[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Multiple)]
internal class Service1 : WcfService1.IService1
{
    public string GetData(int value) => string.Empty;
}

internal class Program
{
    private static void Main(string[] args)
    {
        var baseAddress = new Uri("http://localhost:8080/myservice");
        using (var host = new ServiceHost(typeof(Service1), baseAddress))
        {
            host.Open();
        }
    }
}
```

Fig. 22. Creating and running a WCF service in C#

Figure 23 shows how to create and use a WCF-service client. It connects to an endpoint where the service is operating and calls the service methods.



```
[System.CodeDom.Compiler.GeneratedCode("System.ServiceModel", "4.0.0.0")]
[System.ServiceModel.ServiceContract(ConfigurationName = "ServiceReference1.IService1")]
public interface IService1...

[System.Diagnostics.DebuggerStepThrough()]
[System.CodeDom.Compiler.GeneratedCode("System.ServiceModel", "4.0.0.0")]
public partial class Service1Client ...

internal class Program
{
    private static void Main(string[] args)
    {
        var binding = new BasicHttpBinding();
        var address = new EndpointAddress("http://localhost:8080/myservice");
        var client = new Service1Client(binding, address);
        client.GetData(0);
    }
}
```

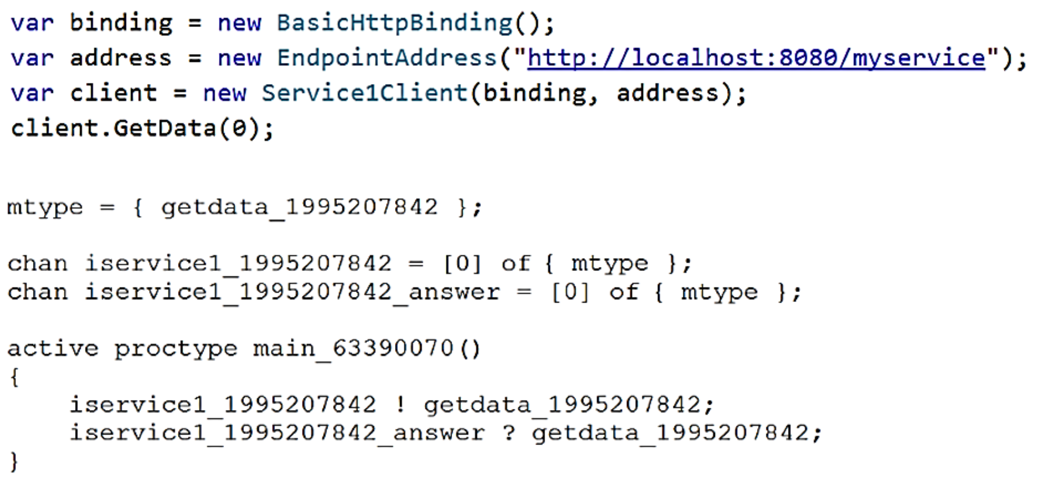
Fig. 23. A way to create and use a WCF client in C#

When modeling the WCF services, here we consider two models – a single-threaded and a multi-threaded server. In C# code both models are implemented by using special annotations (see "ConcurrencyMode=" in Figure 22), in Promela code we model them by generating code to communicate with the client process in the same server process or by using an additional one.

The single-threaded service runs in an infinite loop, receives a message from a client via a rendezvous service channel (buffer size is 0), and there is an indication of a method being called in this message. Then is made a comparison of the requested method with the pre-defined service contract and the execution of the resolved method in the same process which get requests. The multi-threaded service model is similar, except that each handle of the service method occurs in a separate process.

The client model is synchronous. It sends a message to the service channel with the name of the method being called. Then it waits for a response from the service on a separate rendezvous channel.

When generating the model code for the client, we search for an endpoint address it refers and a contract it implements, generate the channels followed by a hash code of the above combination and the messages for service call emulation (Figure 24).



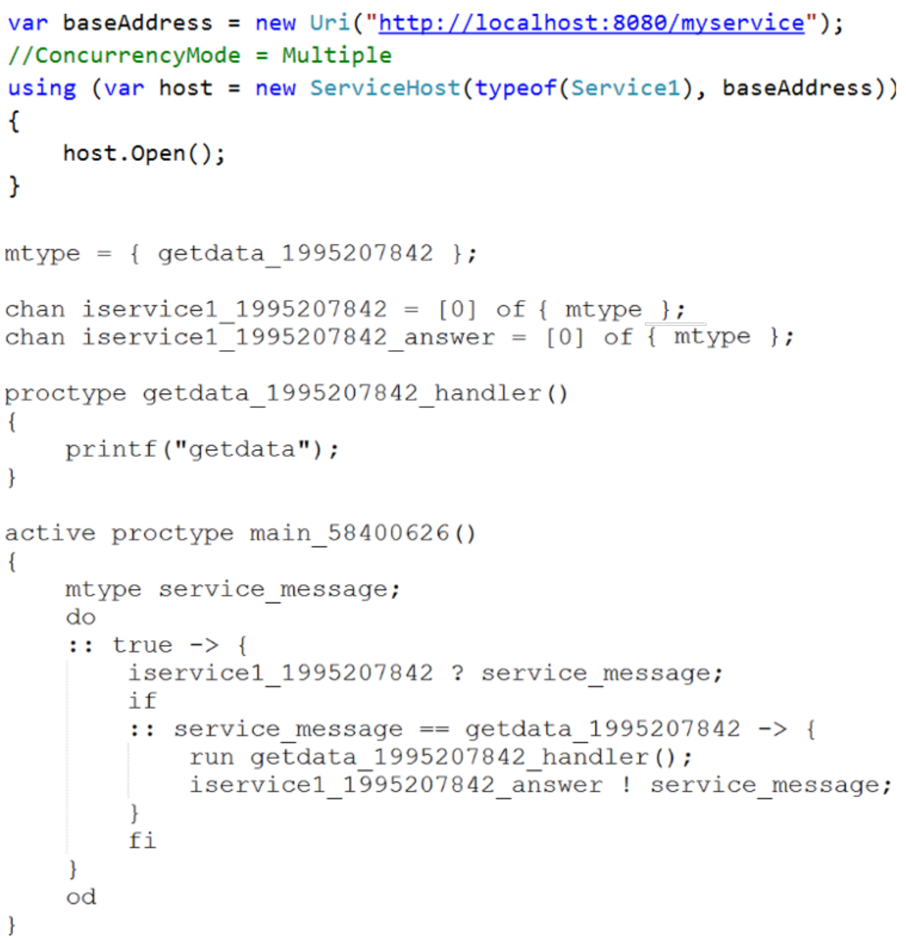
```
var binding = new BasicHttpBinding();
var address = new EndpointAddress("http://localhost:8080/myservice");
var client = new Service1Client(binding, address);
client.GetData(0);
```

```
mtype = { getdata_1995207842 };

chan iservice1_1995207842 = [0] of { mtype };
chan iservice1_1995207842_answer = [0] of { mtype };

active proctype main_63390070()
{
    iservice1_1995207842 ! getdata_1995207842;
    iservice1_1995207842_answer ? getdata_1995207842;
}
```

Fig. 24. Generating Promela code for a WCF service client

When generating the model code for the service, a search for the service contract (the interface to be implemented) is performed to generate the corresponding methods.

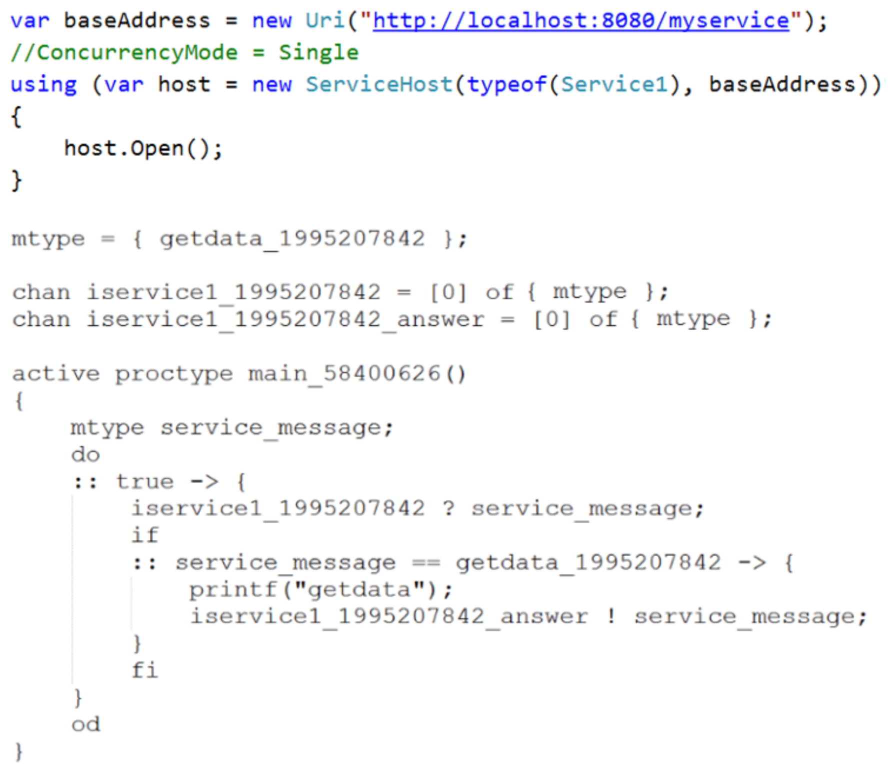
To determine the type of service, the ServiceBehavior attribute of the service class is analyzed. If the argument ConcurrencyMode is set to Multiple, then the multi-threaded model is generated (Figure 25), otherwise the single-threaded one (Figure 26).



```
var baseAddress = new Uri("http://localhost:8080/myservice");
//ConcurrencyMode = Multiple
using (var host = new ServiceHost(typeof(Service1), baseAddress))
{
    host.Open();
}
```

```
mtype = { getdata_1995207842 };

chan iservice1_1995207842 = [0] of { mtype };
chan iservice1_1995207842_answer = [0] of { mtype };

proctype getdata_1995207842_handler()
{
    printf("getdata");
}

active proctype main_58400626()
{
    mtype service_message;
    do
    :: true -> {
        iservice1_1995207842 ? service_message;
        if
        :: service_message == getdata_1995207842 -> {
            run getdata_1995207842_handler();
            iservice1_1995207842_answer ! service_message;
        }
        fi
    }
    od
}
```

Fig. 25. Generating Promela code for a multi-threaded WCF service in C#



```
var baseAddress = new Uri("http://localhost:8080/myservice");
//ConcurrencyMode = Single
using (var host = new ServiceHost(typeof(Service1), baseAddress))
{
    host.Open();
}
```

```
mtype = { getdata_1995207842 };

chan iservice1_1995207842 = [0] of { mtype };
chan iservice1_1995207842_answer = [0] of { mtype };

active proctype main_58400626()
{
    mtype service_message;
    do
    :: true -> {
        iservice1_1995207842 ? service_message;
        if
        :: service_message == getdata_1995207842 -> {
            printf("getdata");
            iservice1_1995207842_answer ! service_message;
        }
        fi
    }
    od
}
```

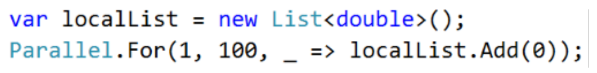
Fig. 26. Generating Promela code for a single-threaded WCF service in C#

4. Verification of distributed system properties

The verification process includes the generation of model code, control variables and rules expressed in LTL, and then model checking (validation of the model correctness).

In this section, we show that by means of generating special control variables and simple LTL formulas, it is possible to model and check some distributed system properties. This process of generation applies to each needed construction as described in the previous section in code independently, so for a particular real program we generate and check different variables and Promela constructions and we do not need to generate complex formulas and structures – we verify the generated model program with respect to all the generated LTL formulas step by step.

4.1. Data race



```
var localList = new List<double>();
Parallel.For(1, 100, _ => localList.Add(0));
```

```
mtype = { thread_complete };
chan awaiter_65946577 = [1] of { mtype }

bool var_458175731_modified = false;
int var_458175731 = 0;

ltl race_conditions_check {
    [] (!var_458175731_modified -> var_458175731 == 0) }

active proctype main_9369539()
{
    do
    :: true -> {
        run lambda_65946577();
    }
    :: true -> break;
    od
}

proctype lambda_65946577()
{
    var_458175731_modified = true;

    var_458175731++;
    var_458175731--;

    var_458175731_modified = false;

    awaiter_65946577 ! thread_complete;
}
```

Fig. 27. A generated Promela code to check the data race using the example of a collection change

The data race problem is the change of object state from different processes (threads) simultaneously that can spoil the correct value of the object.

To model the change we create a pair of control variables $var\_hashcode\_modified = false$ and $var\_hashcode = 0$. The first is set to true before the object is changed, and then to false after the change. The object change itself is modeled using two consecutive operators to increment and decrement the second variable. When the code is correct, this variable should always remain 0.

Thus, data race verification here – is generating and checking the LTL rule (1).

$$G(!var\_hashcode\_modified \rightarrow var\_hashcode == 0) \qquad (1)$$

It means *"always now and in the future, from the fact that the variable does not change, it implies that its reference value is zero"*.

An example of generated Promela code to verify the correct changes of a collection object variable that implements the ICollection <T> interface is shown in Figure 27.

4.2. Improper blocking usage

When modeling a semaphore object, a global shared resource *sem_hashcode_blocked* is used, therefore, when verifying the usage of the semaphore we only need to generate the corresponding LTL rule in the form (2).

$$G(sem\_hash\_code\_blocked \rightarrow FG(!sem\_hash\_code\_blocked)) \qquad (2)$$

It means *"always now and in the future, if the semaphore is locked, then once now or in the future it will be released forever"*.

Note that the rule is stronger than, for example, a rule $G(sem\_hash\_code\_blocked \rightarrow F(!sem\_hash\_code\_blocked))$ because that semaphore object can be used multiple times and we need to ensure that the object finally is stay unlocked if it became locked.

An example of generated Promela code to check the use of a semaphore is shown in Figure 28.

```
var semaphore = new Semaphore(1, 1);
semaphore.WaitOne();
semaphore.Release();
```

```
bool sem__556094084_blocked = false;

ltl sem__556094084_correct {
    [] (sem__556094084_blocked == true -> <>[] (sem__556094084_blocked == false)) }

active proctype main_48266778()
{
    atomic {
        (sem__556094084_blocked == false) -> sem__556094084_blocked = true;
    }
    sem__556094084_blocked = false;
}
```

Fig. 28. Generated Promela code to check a semaphore usage

4.3. Deadlock when accessing an WCF service in a microservice system

Microservices now is a popular concept for scalable [26] applications architectures. Services can act together to solve a huge task by dividing it into some small parts and providing message passing ability between the units. So, a WCF service can be considered as a microservice, if it acts as a service when it receives a task from a client and, in the same time, acts as a client to ask a different service in a chain to get some needed information to the task solving. However, in the chain of service calls, we can get a deadlock when we request a service which is waiting for some data from us. Deadlock can be checked as a liveness condition of the system.

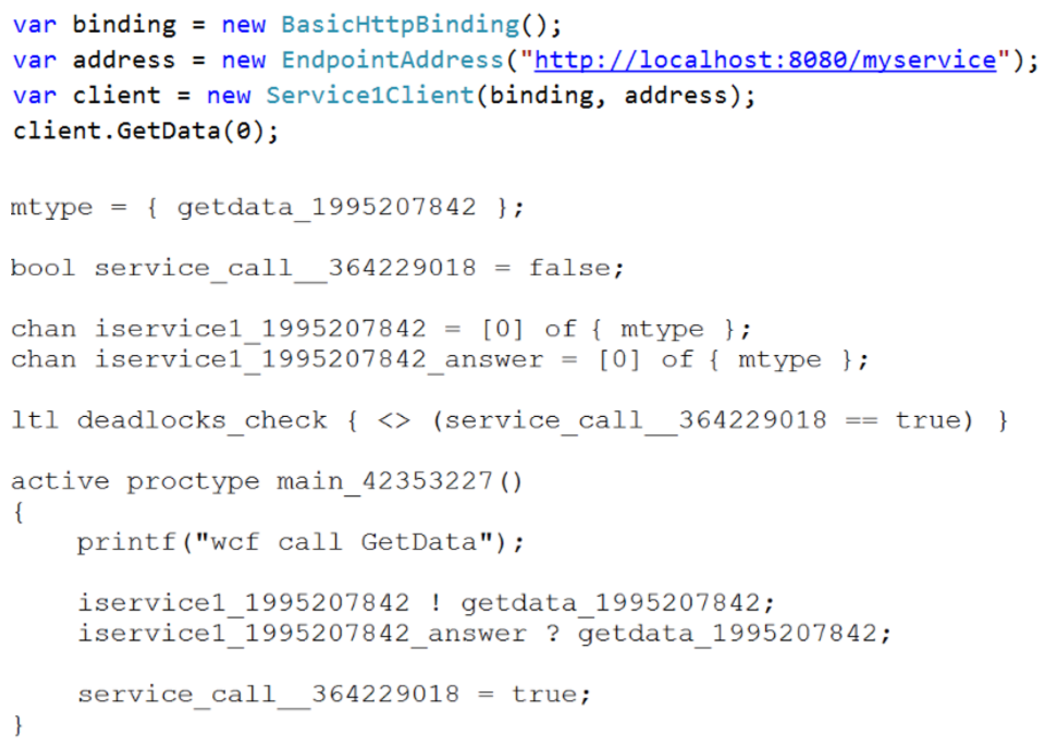
When modeling the service call, we add a separate control variable $service\_call\_hash\_code = false$ before each call to the service method. After receiving a response from the service, we set its value to true.

To verify a deadlock when accessing a service, we generate the LTL rule in the form (3).

$$F(service\_call\_hash\_code == true) \tag{3}$$

It means *"once in the future a response from the service will be received"* (Figure 29).

Using this type of paired constructions of assignments, we ensure that each call to a service function has the corresponding return due to the nature of synchronous calls in WCF. In a microservice chain it is usually hard to capture the logic of corresponding calls (after a call to a service there can exist an inner call to a different service and another next call to another



```
var binding = new BasicHttpBinding();
var address = new EndpointAddress("http://localhost:8080/myservice");
var client = new Service1Client(binding, address);
client.GetData(0);
```

```
mtype = { getdata_1995207842 };

bool service_call__364229018 = false;

chan iservice1_1995207842 = [0] of { mtype };
chan iservice1_1995207842_answer = [0] of { mtype };

ltl deadlocks_check { <> (service_call__364229018 == true) }

active proctype main_42353227()
{
    printf("wcf call GetData");

    iservice1_1995207842 ! getdata_1995207842;
    iservice1_1995207842_answer ? getdata_1995207842;

    service_call__364229018 = true;
}
```

Fig. 29. Generated Promela code to check for a deadlock when accessing a WCF service

service and so on), and by expecting a call return for all calls we can check the correctness of the whole system and easily find a place where the integrity of the service calls might violate.

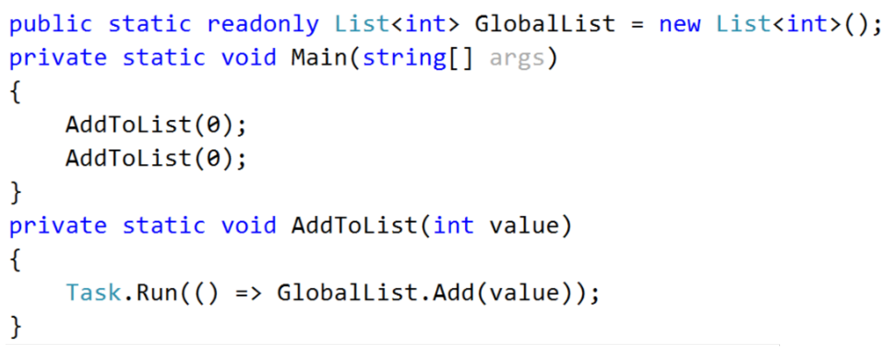
5. An example of generated model verification

Consider an example of starting and running SPIN verifier with a generated model and an LTL formula that checks it for the data race.

An example source code in C# and a corresponding generated model in Promela are shown in Figure 30.

Figure 31 shows the result of the verification signaling the violation of the absence of data race condition.

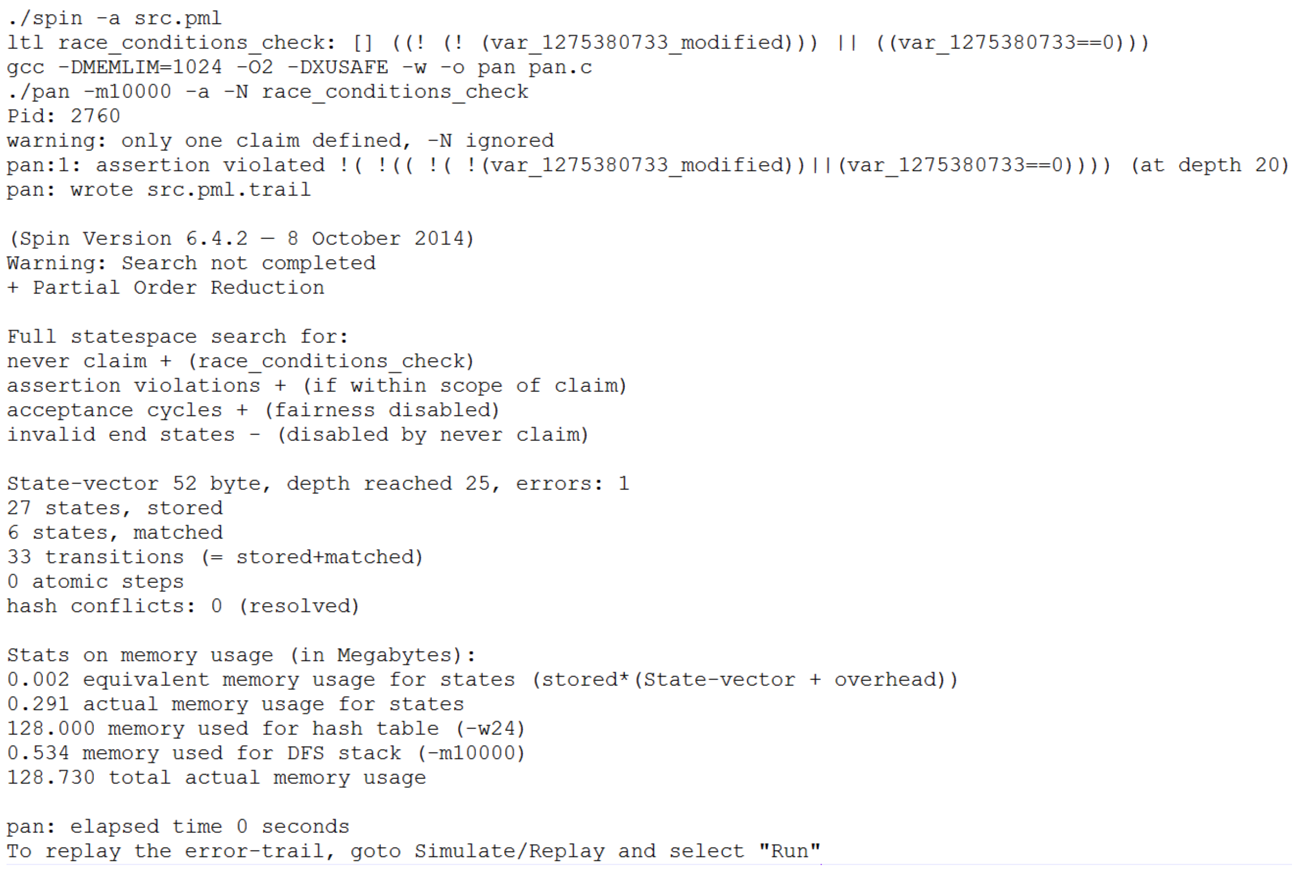
Since the verifier found a counterexample, we can build a call trail containing it in the simulation mode with SPIN (Figure 32).



```
public static readonly List<int> GlobalList = new List<int>();
private static void Main(string[] args)
{
    AddToList(0);
    AddToList(0);
}
private static void AddToList(int value)
{
    Task.Run(() => GlobalList.Add(value));
}
```

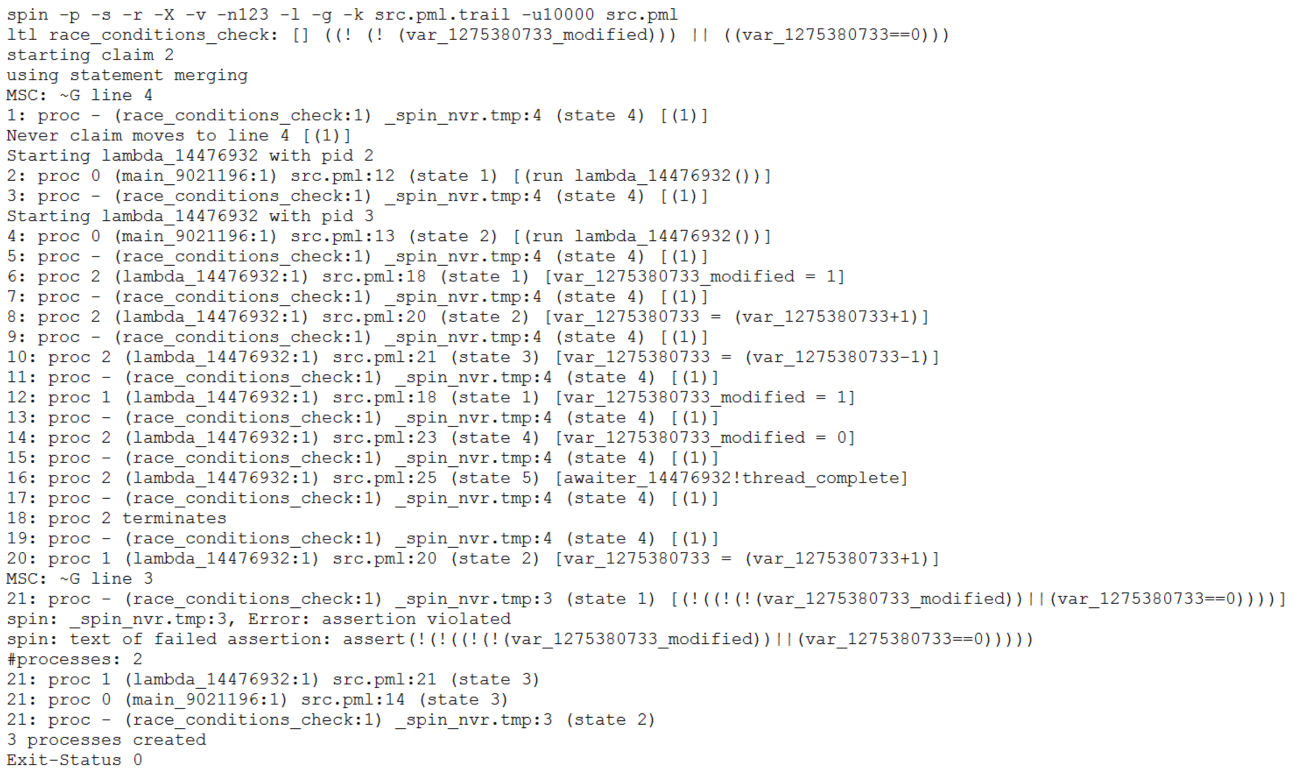
```
mtype = { thread_complete };

chan awaiter_66898905 = [1] of { mtype }

bool var_31747823_modified = false;
int var_31747823 = 0;

ltl race_conditions_check {
    [] (!var_31747823_modified -> var_31747823 == 0) }

active proctype main_20561848()
{
    run lambda_66898905();
    run lambda_66898905();
}

proctype lambda_66898905()
{
    var_31747823_modified = true;
    var_31747823++;
    var_31747823--;
    var_31747823_modified = false;
    awaiter_66898905 ! thread_complete;
}
```

Fig. 30. Generated source code for checking the race condition



```
./spin -a src.pml
ltl race_conditions_check: [] ((! (! (var_1275380733_modified))) || ((var_1275380733==0)))
gcc -DMEMLIM=1024 -O2 -DXUSAFE -w -o pan pan.c
./pan -m10000 -a -N race_conditions_check
Pid: 2760
warning: only one claim defined, -N ignored
pan:1: assertion violated !( !(( !( !(var_1275380733_modified))||(var_1275380733==0)))) (at depth 20)
pan: wrote src.pml.trail

(Spin Version 6.4.2 — 8 October 2014)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never claim + (race_conditions_check)
assertion violations + (if within scope of claim)
acceptance cycles + (fairness disabled)
invalid end states - (disabled by never claim)

State-vector 52 byte, depth reached 25, errors: 1
27 states, stored
6 states, matched
33 transitions (= stored+matched)
0 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
0.002 equivalent memory usage for states (stored*(State-vector + overhead))
0.291 actual memory usage for states
128.000 memory used for hash table (-w24)
0.534 memory used for DFS stack (-m10000)
128.730 total actual memory usage

pan: elapsed time 0 seconds
To replay the error-trail, goto Simulate/Replay and select "Run"
```

Fig. 31. Verification result indicating the violation of the absence of data race condition



```
spin -p -s -r -X -v -n123 -l -g -k src.pml.trail -u10000 src.pml
ltl race_conditions_check: [] ((! (! (var_1275380733_modified))) || ((var_1275380733==0)))
starting claim 2
using statement merging
MSC: ~G line 4
1: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
Never claim moves to line 4 [(1)]
Starting lambda_14476932 with pid 2
2: proc 0 (main_9021196:1) src.pml:12 (state 1) [(run lambda_14476932())]
3: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
Starting lambda_14476932 with pid 3
4: proc 0 (main_9021196:1) src.pml:13 (state 2) [(run lambda_14476932())]
5: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
6: proc 2 (lambda_14476932:1) src.pml:18 (state 1) [var_1275380733_modified = 1]
7: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
8: proc 2 (lambda_14476932:1) src.pml:20 (state 2) [var_1275380733 = (var_1275380733+1)]
9: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
10: proc 2 (lambda_14476932:1) src.pml:21 (state 3) [var_1275380733 = (var_1275380733-1)]
11: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
12: proc 1 (lambda_14476932:1) src.pml:18 (state 1) [var_1275380733_modified = 1]
13: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
14: proc 2 (lambda_14476932:1) src.pml:23 (state 4) [var_1275380733_modified = 0]
15: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
16: proc 2 (lambda_14476932:1) src.pml:25 (state 5) [awaiter_14476932!thread_complete]
17: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
18: proc 2 terminates
19: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
20: proc 1 (lambda_14476932:1) src.pml:20 (state 2) [var_1275380733 = (var_1275380733+1)]
MSC: ~G line 3
21: proc - (race_conditions_check:1) _spin_nvr.tmp:3 (state 1) [(!((!(!(var_1275380733_modified))||(var_1275380733==0))))]
spin: _spin_nvr.tmp:3, Error: assertion violated
spin: text of failed assertion: assert(!(!((!(!(var_1275380733_modified))||(var_1275380733==0)))))
#processes: 2
21: proc 1 (lambda_14476932:1) src.pml:21 (state 3)
21: proc 0 (main_9021196:1) src.pml:14 (state 3)
21: proc - (race_conditions_check:1) _spin_nvr.tmp:3 (state 2)
3 processes created
Exit-Status 0
```

Fig. 32. Counterexample in the simulation mode

6. Conclusion

As the result of the work, we created a possible extension of the C# language compiler toolset for the formal verification of parallel and distributed applications, which:

- transforms a C# program into an actor model in Promela;
- creates control variables;
- generates LTL formulas for checking:
  - deadlocks;
  - data races;
  - synchronization errors;
- verifies the generated model according to the generated rules.

The aim of the paper was not to create a complete solution for proving parallel and distributed software, we just wanted to show a possible method that uses AST C# code model to Promela model transformation, generates LTL formulas and checks them with the SPIN verifier.

The process is based on analyzing Roslyn structures of full C# grammar programs. The results can be used in static checkers to prove some properties of sophisticated code.

The method we used can be a bridge from real programs to its formal models with the automatic transformation between them.

Bibliography

1. Clarke Jr, Edmund M., et al. Model checking. Cyber-Physical Systems, 2018.
2. Staroletov, Sergey. Basics of Software Testing and Verification [in Russian]. Lanbook, Saint Petersburg, 2018. -344p. ISBN 978-5-8114-3041-3.
3. Richter, Jeffrey. CLR via C#. Pearson Education, 2012. ISBN 978-0-7356-6876-8
4. Home repository for .NET Core. Available from: https://github.com/dotnet/core
5. McMurtry, Craig, et al. "Windows Communication Foundation Unleashed (WCF)." (2007).
6. The Roslyn .NET compiler provides C# and Visual Basic languages with rich code analysis APIs. Available from: https://github.com/dotnet/roslyn
7. Harrison, Nick. Code Generation with Roslyn. Apress, 2017. ISBN 978-1-4842-2211-9
8. Vasiliev, Sergey. Introduction to Roslyn and its use in program development. Available from: *https : //www.viva64.com/en/b/0399/*
9. Burghardt, Jochen, et al. "ACSL By Example." (2016).
10. Hewitt, Carl, Peter Bishop, and Richard Steiger. "Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence." Advance Papers of the Conference. Vol. 3. Stanford Research Institute, 1973.
11. Agha, Gul A. Actors: A model of concurrent computation in distributed systems. No. AI-TR-844. Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1985.
12. Armstrong, Joe. Programming Erlang: software for a concurrent world. Pragmatic Bookshelf, 2013.
13. Vernon, Vaughn. Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka. Addison-Wesley Professional, 2015.
14. Holzmann, Gerard J. "The model checker SPIN." IEEE Transactions on software engineering 23.5 (1997): 279-295.
15. Pnueli, Amir. "The temporal logic of programs." Foundations of Computer Science, 1977., 18th Annual Symposium on. IEEE, 1977.
16. Proctype – for declaring new process behavior. Promela. Available from: *http : //spinroot.com/spin/Man/proctype.html*
17. Michael Barnett, K. Rustan, M. Leino and Wolfram Schulte. "The Spec# programming system: An overview." International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. Springer, Berlin, Heidelberg, 2004.
18. Manuel Fähndrich, Michael Barnett and Francesco Logozzo. "Embedded contract lan-

guages." Proceedings of the 2010 ACM Symposium on Applied Computing. ACM, 2010.

19. Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. Proceedings of EuroSys2006. Leuven, Belgium, April 2006. ACM SIGOPS.
20. Koshelev, Vladimir Konstantinovich, Valery Nikolayevich Ignatyev, and Artem Il'ich Borzilov. "C# static analysis framework." Proceedings of the Institute for System Programming of the RAS 28.1 (2016): 21-40.
21. Karpov, Andrey. How PVS-Studio does the bug search: methods and technologies. Available from: https://www.viva64.com/en/b/0466/
22. Staroletov, Sergey. Model of a program as multi-threaded stochastic automaton and its equivalent transformation to Promela model. Ershov informatics conference. PSI Series, 8th edition. International workshop on Program Understanding. Proceedings. – Novosibirsk, 2011. p. 33-38
23. Gamma, Erich, et al. "Design patterns: Abstraction and reuse of object-oriented design." European Conference on Object-Oriented Programming. Springer, Berlin, Heidelberg, 1993.
24. Inline – a stylized version of a macro. Promela. Available from: *http://spinroot.com/spin/Man/inline.html*
25. Do – repetition construct. Promela. Available from: *http://spinroot.com/spin/Man/do.html*
26. Dragoni, Nicola, et al. "Microservices: How to make your application scale." International Andrei Ershov Memorial Conference on Perspectives of System Informatics. Springer, Cham, 2017.